**ASSIGNMENT QUESTION 10 : SMART BRIDGE LOAD MONITOR**

NAME : Mani Hasini

ROLL NUMBER : ME24B1067

DATE : 15-04-2025

PROBLEM STATEMENT

Here, I am designing a smart system to monitor and maintain a bridge's structural health. Load tests from sensors are queued, critical stress alerts are prioritized using a stack, and results are logged in a limited-capacity history unit with archiving. Components showing stress or damage are tracked through linked lists based on their maintenance status—stressed, reinforced, or urgent repair. The goal is to simulate this real-time safety and maintenance process efficiently.

OBJECTIVE

The objective of this C program is to simulate a smart bridge load monitoring system that manages real-time load test data, detects critical stress points, maintains a history of test results, and tracks maintenance activities using appropriate data structures. It uses a queue for managing load requests, a stack for urgent stress checks, an array for logging results with archiving, and various linked lists to track stressed, reinforced, and high-priority components for repair and review.

DESIGN EXPLANATION

This system mimics how a smart bridge would handle safety checks in real life. Load tests from sensors line up in a **queue**, and critical stress points get checked urgently using a **stack**. Test results are saved in a **history log (array)**, where older ones are archived once full. If any parts show stress, they’re added to a **stressed list (singly linked list)**. Reinforced ones move to a **review list (doubly linked list)**, and urgent repairs like cracks are tracked in a **circular list** for constant monitoring.

CODE LOGIC AND IMPLEMENTATION

Step 1: Load Test Input (Queue)

* Sensors send load test requests (e.g., "Beam", "Cable", etc.).
* These are added to a **queue**, ensuring they’re handled in the order they arrive (FIFO).

Step 2: Stress Check (Stack)

* Each component is removed from the queue and **pushed onto a stack**.
* The stack lets us check the **most recent components first** (LIFO), which is useful in emergency checks.

Step 3: Result Logging (Circular Array)

* Every test result is saved in a **6-slot array**.
* If more than 6 results come in, the oldest one is **archived** (overwritten), and the newest is stored.

Step 4: Stressed Components Tracking (Singly Linked List)

* If a component (like "Cable" or "Joint") is found to be under stress, it's added to a **singly linked list**.
* This helps track which parts still need attention.

Step 5: Reinforced Components Review (Doubly Linked List)

* Once a stressed component (like "Cable") is fixed, it's moved to a **doubly linked list**.
* This allows **forward and backward review** of reinforced parts during inspections.

Step 6: Urgent Repairs (Circular Linked List)

* Components like "Beam" and "Pylon" needing fast or repeated repairs are added to a **circular list**.
* This structure keeps **looping through urgent items**, ensuring they’re never missed.

**CODE :**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define SIZE 6 // For load history

// Structures

typedef struct Node {

char data[20];

struct Node\* next;

struct Node\* prev; // Only used in DLL

} Node;

// Queue

typedef struct {

char data[6][20];

int front, rear;

} Queue;

// Stack

typedef struct {

char data[6][20];

int top;

} Stack;

// Circular Linked List Node (reuse Node but with circular next)

Node\* circularHead = NULL;

// Queue Functions

void initQueue(Queue\* q) {

q->front = q->rear = -1;

}

void enqueue(Queue\* q, char\* item) {

if (q->rear == 5) {

printf("Queue Full!\n");

return;

}

q->rear++;

strcpy(q->data[q->rear], item);

if (q->front == -1) q->front = 0;

}

char\* dequeue(Queue\* q) {

if (q->front == -1) return NULL;

char\* item = q->data[q->front];

if (q->front == q->rear) q->front = q->rear = -1;

else q->front++;

return item;

}

// Stack Functions

void initStack(Stack\* s) {

s->top = -1;

}

void push(Stack\* s, char\* item) {

if (s->top == 5) {

printf("Stack Overflow!\n");

return;

}

strcpy(s->data[++s->top], item);

}

char\* pop(Stack\* s) {

if (s->top == -1) return NULL;

return s->data[s->top--];

}

// Load History

char loadHistory[SIZE][20];

int start = 0, count = 0;

void logResult(char\* result) {

if (count < SIZE) {

strcpy(loadHistory[(start + count) % SIZE], result);

count++;

} else {

printf("Archiving: %s\n", loadHistory[start]);

strcpy(loadHistory[start], result);

start = (start + 1) % SIZE;

}

}

// Linked List Helpers

Node\* createNode(char\* data) {

Node\* newNode = (Node\*)malloc(sizeof(Node));

strcpy(newNode->data, data);

newNode->next = newNode->prev = NULL;

return newNode;

}

// Singly Linked List

Node\* stressedHead = NULL;

void insertSLL(char\* data) {

Node\* newNode = createNode(data);

newNode->next = stressedHead;

stressedHead = newNode;

}

void deleteSLL(char\* data) {

Node\* temp = stressedHead, \*prev = NULL;

while (temp && strcmp(temp->data, data) != 0) {

prev = temp;

temp = temp->next;

}

if (!temp) return;

if (!prev) stressedHead = temp->next;

else prev->next = temp->next;

free(temp);

}

// Doubly Linked List

Node\* reinforcedHead = NULL, \*reinforcedTail = NULL;

void insertDLL(char\* data) {

Node\* newNode = createNode(data);

if (!reinforcedHead) {

reinforcedHead = reinforcedTail = newNode;

} else {

reinforcedTail->next = newNode;

newNode->prev = reinforcedTail;

reinforcedTail = newNode;

}

}

void traverseDLL() {

Node\* temp = reinforcedHead;

printf("Forward (DLL): ");

while (temp) {

printf("%s ", temp->data);

temp = temp->next;

}

printf("\nBackward (DLL): ");

temp = reinforcedTail;

while (temp) {

printf("%s ", temp->data);

temp = temp->prev;

}

printf("\n");

}

// Circular Linked List

void insertCLL(char\* data) {

Node\* newNode = createNode(data);

if (!circularHead) {

circularHead = newNode;

circularHead->next = circularHead;

} else {

Node\* temp = circularHead;

while (temp->next != circularHead) temp = temp->next;

temp->next = newNode;

newNode->next = circularHead;

}

}

void traverseCLL(int times) {

if (!circularHead) return;

Node\* temp = circularHead;

printf("Circular List Traversal: ");

for (int i = 0; i < times \* 2; i++) {

printf("%s ", temp->data);

temp = temp->next;

}

printf("\n");

}

// Main Simulation

int main() {

Queue q;

Stack s;

initQueue(&q);

initStack(&s);

// (a) Load Tests → Queue → Stack → Pop

char\* components[] = {"Beam", "Cable", "Deck", "Pylon", "Joint", "Bolt"};

printf("Enqueuing Load Tests:\n");

for (int i = 0; i < 6; i++) {

enqueue(&q, components[i]);

printf("- %s\n", components[i]);

}

printf("\nPushing to Stress Stack:\n");

while (q.front != -1) {

char\* item = dequeue(&q);

push(&s, item);

printf("Pushed: %s\n", item);

}

printf("\nStress Check Order (LIFO):\n");

char\* stress;

while ((stress = pop(&s)) != NULL) {

printf("- Check: %s\n", stress);

}

// (b) Load History

printf("\nLogging Load History:\n");

for (int i = 1; i <= 8; i++) {

char result[10];

sprintf(result, "Res%d", i);

logResult(result);

}

printf("Current Load History:\n");

for (int i = 0; i < count; i++) {

printf("- %s\n", loadHistory[(start + i) % SIZE]);

}

// (c) Stressed Components

printf("\nTracking Stressed Components:\n");

insertSLL("Cable");

insertSLL("Joint");

printf("Reinforcing Cable...\n");

deleteSLL("Cable");

insertDLL("Cable");

traverseDLL();

// (d) Urgent Repair Cycle

printf("\nAdding Urgent Repairs:\n");

insertCLL("Beam");

insertCLL("Pylon");

traverseCLL(2); // Traverse twice

return 0;

}

**SAMPLE OUTPUT :**

Enqueuing Load Tests:

- Beam

- Cable

- Deck

- Pylon

- Joint

- Bolt

Pushing to Stress Stack:

Pushed: Beam

Pushed: Cable

Pushed: Deck

Pushed: Pylon

Pushed: Joint

Pushed: Bolt

Stress Check Order (LIFO):

- Check: Bolt

- Check: Joint

- Check: Pylon

- Check: Deck

- Check: Cable

- Check: Beam

Logging Load History:

Archiving: Res1

Archiving: Res2

Current Load History:

- Res3

- Res4

- Res5

- Res6

- Res7

- Res8

Tracking Stressed Components:

Reinforcing Cable...

Forward (DLL): Cable

Backward (DLL): Cable

Adding Urgent Repairs:

Circular List Traversal: Beam Pylon Beam Pylon